Introduction

In this project, the goal is to solve a logical expression in CNF form. In fact, 429 statements are given, all of which must be correct. These logical expressions must be defined in the form of 100 variables that take the value of zero and one.

Each entry contains 100 zeros and ones.

At first, we read the logical expression and save it as a presentation. The array is two-dimensional and has 429 members and each member is three.

The program works by receiving an input:

[1]: genetics

[2]: simulated

Enter your choice:

We enter the number 1 or 2 and then the corresponding algorithms are executed. If a wrong input is given to the program, the following statement will be displayed:

Please enter 1 or 2

The two functions that are used in both common algorithms are:

- init_state
- count_incorrect_caluses

The first function is used to create a random input. For example, it creates an array of 100 as follows:

[0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 , 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0 , 0, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1 , 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 0]

The second function checks how many of the 429 statements with one input are not correct. And finally, each score

The solution is calculated as the difference between 429 and the number reported by this function.

Genetic Algorithm This algorithm is defined in genetic function. The function first creates a random response using state_init and defines the following variables:

Number of repetitions: iteration

Mutation rate: rate

Initial population: k

For each repetition, fitness and maximum score are calculated. If the score reaches 429, it means that the answer has been found.

```
f = f + (len(clauses) - count_incorrect_caluses(clauses[:],population[0]))/float(len(clauses))

fitness = f/len(population)

best_score = (len(clauses)-
count_incorrect_caluses(clauses[:],population[0]))/float(len(clauses))
```

Then it's time to build a new population. In this case, a clone is first taken from the primary population, which is defined based on the chance based on the score, the higher the score, the higher the chance.

```
for i in range(k):

        chances=chances+([i]* int((len(clauses)-

count_incorrect_caluses(clauses[:],population[i]))/10))

for i in range(k):

        cloned_population.append(population[choice(chances)])
```

In the next section, the previous population and the new population are crossed:

```
for i in range(k):

        limit=randint(0,n_variables-1)

 new_population[i]=cloned_population[randint(0,k1)][:limit]+cloned_population[randint(0,k-
1)][limit:]
```

In the last part, it is the turn of mutation. Using a new population, the mutation rate of the population is created.

```
 krate=random()%rate

 limit=int(len(new_population)*krate)

 for i in range(limit):

        r=randint(0,n_variables-1)

        new_population[randint(2,k-1)][r]=new_population[i][r]*-1
```

Thermal simulation algorithm

Thermal simulation algorithm is implemented by simulated function.

The function first creates a random response using state_init and defines the following variables:

- Number of iterations: iterations_n
- The number of variables equal to 100: variables_n
- The initial value of temperature is 5. temperature

For each repetition, a random solution is generated and its score is calculated. If the score is higher than the previous solution, this solution will be replaced with temperature-dependent probability:

curr = [1-var if random() < 0.01 else var for var in solution]

delta = best_eval - curr_eval

if delta <= 0 or random() < np.exp(-delta/temperature):

solution = curr

best_eval = curr_eval

Also, the temperature is updated as follows:

temperature = (1 - (i+1)/n_iterations)

Finally, this algorithm reaches 95-97% accuracy.